An Overview of Program Optimization Techniques

accu

professionalism in programming

Mathias Gaunard

Maven Securities

April 27, 2017

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Program Optimization

Designing Optimized Programs

- Process big chunks of work faster
- React faster to certain events
- Use less resources

An Open Topic

- Compromises need to be made depending on what you want to optimize
- Difficult to apply to complex systems due to all variables involved
- “Poking at things in the dark”
  - □ Micro-benchmarks or fine measurements introduce bias
  - □ Static analysis doesn’t have all the data
  - □ Only good measure is the whole system performance in production

An Overview

Goals of this Talk

- A non-exhaustive number of topics that affect performance
- High-level description rather than in-depth analysis
- A few recipes around those topics
- Optimization is fun, but often evil
  - □ Don’t over-engineer everyday tasks
  - □ Investigate algorithmic complexity first
  - □ Experiment, measure and optimize responsibly

Talk Structure

Part 1: Understanding the Hardware

- How source code is mapped to a micro-architecutre
- What the chip does to speed up your code
- How the chip interacts with the outside world, aka main memory

Part 2: Writing Code for the Hardware

- Typical patterns and source code transformations
- Some C++ recipes and tools

Outline

Introduction

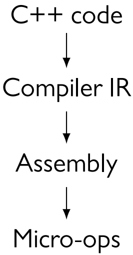
Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Mapping C++ to the Hardware



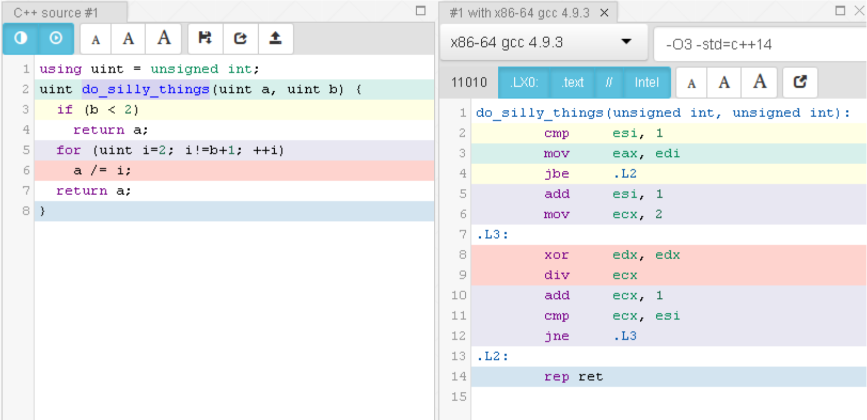
C++ code

Compiler IR

Assembly

Micro-ops

C++ to Assembly



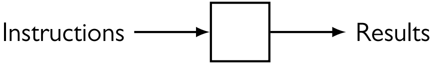
-O3 -std=c++14

```cpp
using uint = unsigned int;
uint do_silly_things(uint a, uint b) {
  if (b < 2)
    return a;
  for (uint i=2; i!=b+1; ++i)
    a /= i;
  return a;
}
```

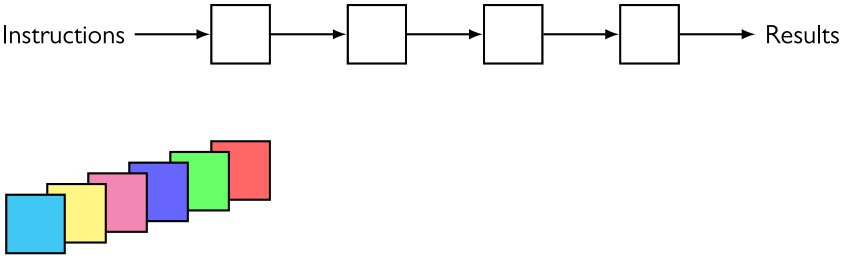
```
do_silly_things(unsigned int, unsigned int):
        cmp     esi, 1
        mov     eax, edi
        jbe     .L2
        add     esi, 1
        mov     ecx, 2
.L3:
        xor     edx, edx
        div     ecx
        add     ecx, 1
        cmp     ecx, esi
        jne     .L3
.L2:
        rep ret
```

Figure: x86-64 assembly of simple C++ code, courtesy of gcc.godbolt.org

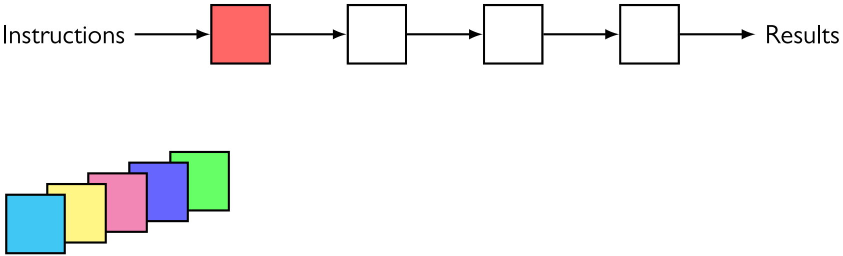
What a CPU does



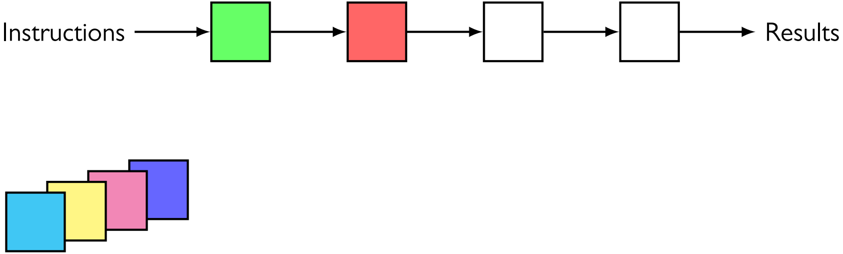
What a CPU does: Pipelining



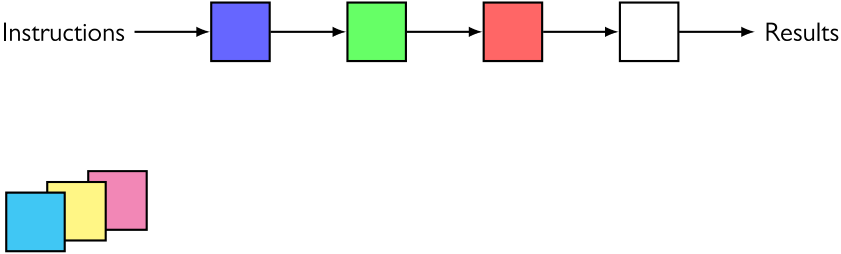
What a CPU does: Pipelining



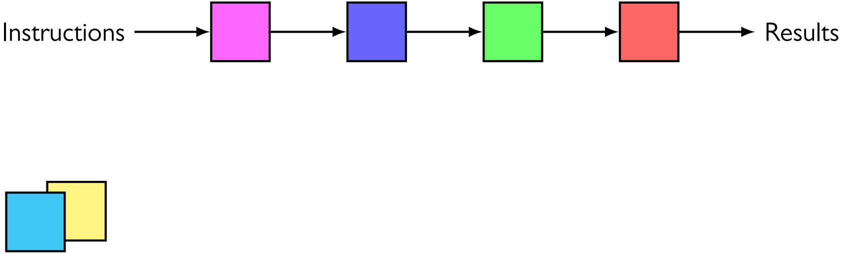
What a CPU does: Pipelining



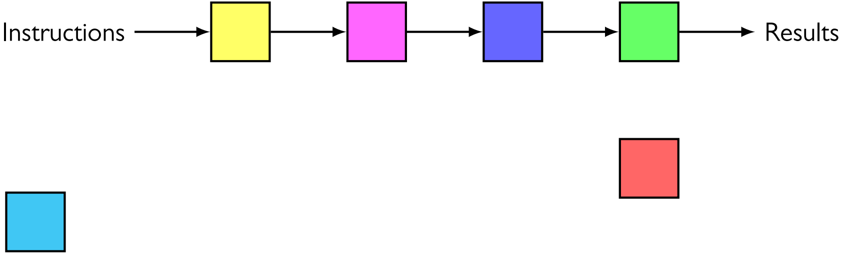
What a CPU does: Pipelining



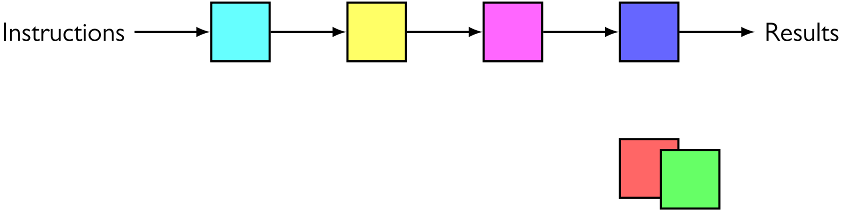
What a CPU does: Pipelining



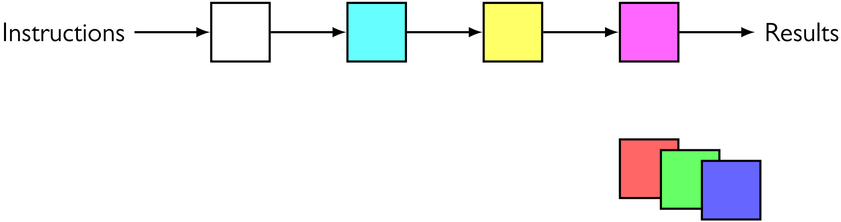
What a CPU does: Pipelining



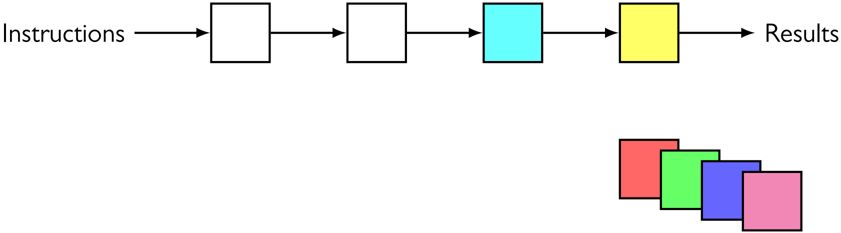
What a CPU does: Pipelining



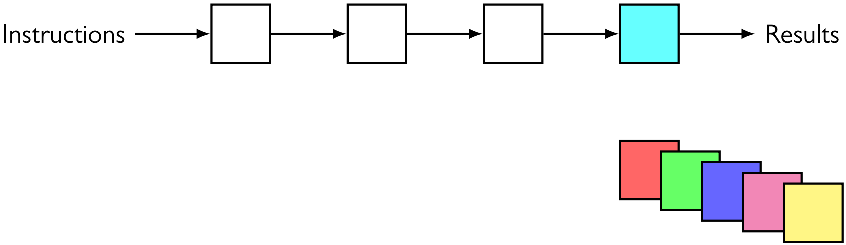
What a CPU does: Pipelining



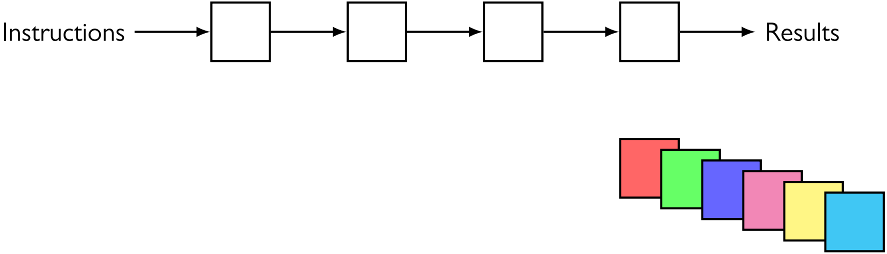
What a CPU does: Pipelining



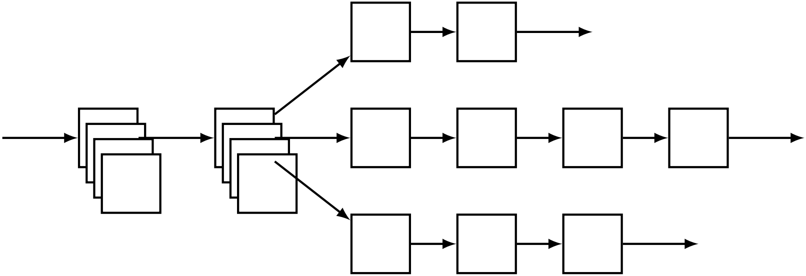
What a CPU does: Pipelining



What a CPU does: Pipelining



What a CPU does: Superscalar



A Superscalar Pipeline

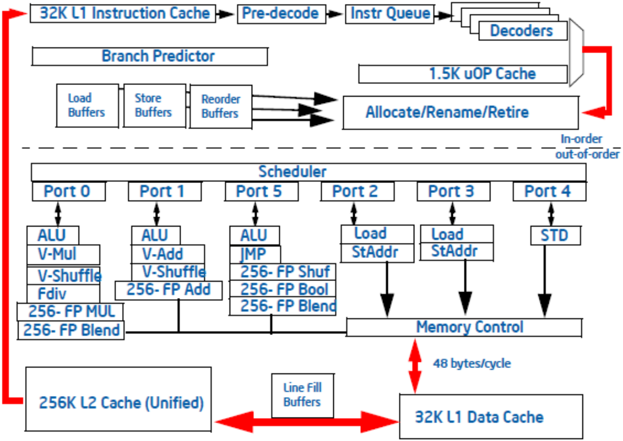


Figure: Intel SandyBridge micro-architecture

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Stalls in the Pipeline

Imaginary Instructions

- load/store 3 cycles
- add 1 cycle

Stalls (15 cycles)

```
r1 = load(mem[i]);
r1 = add(r1, 2);
store(r1, mem[j])
r1 = load(mem[i+1])
r1 = add(r2, 2)
store(r1, mem[j+1])
r1 = load(mem[i+2])
r1 = add(r23 3)
store(r1, mem[j+2])
```

No stalls (9 cycles)

```
r1 = load(mem[i]);
r2 = load(mem[i+1])
r3 = load(mem[i+2])
r1 = add(r1, 2);
r2 = add(r2, 2)
r3 = add(r3, 2)
store(r1, mem[j])
store(r2, mem[j+1])
store(r3, mem[j+2])
```

Superscalar Execution

Multiple Execution Ports

- Some operations available on multiple ports, some only on some
- Different ports have different types of processing units
  - □ ALU
  - □ FPU
  - □ Memory Control
  - □ Special Instructions

Port Details

- Dynamically scheduled as new instructions are evaluated
- Each port has its own internal pipeline
- Some static analysis tools exist to deduce port allocation for Intel

Out-of-Order Execution

In-Order Execution

- If an instruction depends on the result of a previous instruction, the processor must wait until it retires before executing (stall)
- Requires statically scheduling instructions for efficiency

Out-of-Order Execution

- Instructions are queued and executed as their dependencies become available
- Dynamic scheduling method
- No sequential consistency, various relaxed ordering models
- x86: total store order consistency
  - reads not reordered with other reads
  - writes not reordered with other writes
  - writes can be reordered after reads
  - reads can be reordered after writes (if not dependent)

Working with a OoO Superscalar Processor

Multi-Issue

- Processors can actually issue multiple instructions per cycle
- Study reference manuals (latency, throughput, port mappings) to see what can be executed concurrently to maximize CPU utilization

Avoiding Stalls

- Learn to identify them with profiling tools
- Avoid dependencies that prevent re-ordering, such as read after write
- Pipelining manually can still help

Specialized Instructions

Not All Instructions Equal

- Some instructions handled by more ports than others
- Even then, some instructions faster than others
- Addition is fastest, multiplication fast, division slow
- Single precision faster than double precision

Instruction Set Architecture Extensions

- Special bit manipulation instructions
- Crypography instructions
- SIMD processing units

Relative Instruction Speed

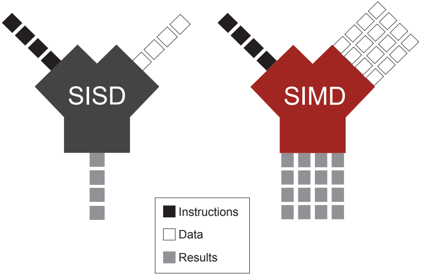
Fast to Slow

- tests
- bitwise operations, integer add
- floating-point add
- integer multiplication
- floating-point multiplication
- floating-point division (slow!)
- integer division (super slow!)

YMMV

- slower instructions have a higher *latency*
- high *throughput* can still be obtained with pipelining

SIMD



Principles

- Single Instruction, Multiple Data
- Large register with multiple lanes, each operation applied to all lanes
- If 128-bit register, processes 4 32-bit values or 2 64-bit values in parallel

SIMD Assembly

General-purpose x86-64 Assembly

```
    xor      eax, eax
    cmp      rdi, rsi
    je       .L4
.L3:
    movzx    edx, WORD PTR [rdi]
    add      rdi, 2
    add      eax, edx
    cmp      rsi, rdi
    jne      .L3
.L4:
```

SSE2 Assembly

```
    cmp          rdi, rsi
    je           .L15
    pxor         xmm2, xmm2
    pxor         xmm3, xmm3
.L55:
    movdqa       xmm0
, XMMWORD PTR [rdi]
    add          rdi, 16
    cmp          rsi, rdi
    movdqa       xmm4, xmm0
    movdqa       xmm1, xmm0
    punpcklwd    xmm4, xmm3
    punpckhwd    xmm1, xmm3
    movdqa       xmm0, xmm4
    paddd        xmm0, xmm2
    paddd        xmm0, xmm1
    movdqa       xmm2, xmm0
    jne          .L55
.L15:
```

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Simultaneous Multi-Threading

Temporal Multi-Threading

- Classical approach for multithreading on non-parallel processors
- Time sharing, one thread at a time

SMT: An Improvement for Superscalar Processors

- Executes concurrently multiple threads on the same superscalar core
- Shares the processing units across the threads, maximizing port occupation
- Can lead to resource starvation and decreased efficiency per core
- Also known as HyperThreading

SMP and NUMA

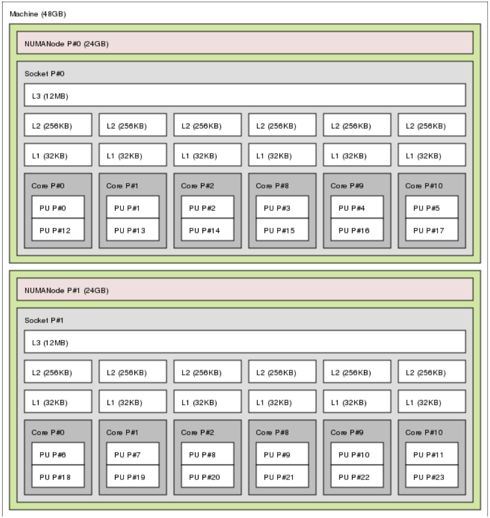
Shared Memory Parallelism

- Multiple cores on a single socket (multi-core)
- Multiple socket on a single bus (Symmetric Multi-Processing)
- Multiple nodes inter-connected together (Non-Uniform Memory Access)

Hierarchical Affinity

- Some cores closer to each other, cheaper communication
- Some local memory
- Some shared memory
- Some remote shared memory

NUMA Architecture



Machine (48GB)

NUMANode P#0 (24GB)

Socket P#0

L3 (12MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)
Core P#0	Core P#1	Core P#2	Core P#8	Core P#9	Core P#10
PU P#0	PU P#1	PU P#2	PU P#3	PU P#4	PU P#5
PU P#12	PU P#13	PU P#14	PU P#15	PU P#16	PU P#17

NUMANode P#1 (24GB)

Socket P#1

L3 (12MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)	L1 (32KB)
Core P#0	Core P#1	Core P#2	Core P#8	Core P#9	Core P#10
PU P#6	PU P#7	PU P#8	PU P#9	PU P#10	PU P#11
PU P#18	PU P#19	PU P#20	PU P#21	PU P#22	PU P#23

Memory Allocation on NUMA

Page Granularity

- Page can be allocated on any of the NUMA nodes
- Mapped into virtual address space at any location
- Typically never migrates once allocated

First Touch

- Linux over-commits, allocates page when data is actually written to it
- Policy is to allocate on the node where the code is being run
- Data should be initialized or copied by each worker rather than a master

Effects of Multithreading

Context Switches

- Context switching is a potentially expensive operation (∼1000 cycles)
- Going to the kernel causes a context switch
- The kernel will schedule and switch to other threads to the core
- One given thread might move to other cores during its lifetime, losing some cached content

Cache Effects

- If a core writes to a cache line, other cores need to evict it
- False sharing if multiple cores access independent data on the same cache line

Synchronization

Mutexes

- Put thread to sleep and do something else until condition is satisfied
- Costly but better resource management

Memory Barriers

- Prevents out-of-ordering re-ordering
- Combined with atomicity, allows lightweight synchronization
- Can cause more stalls to occur

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Memory Technology

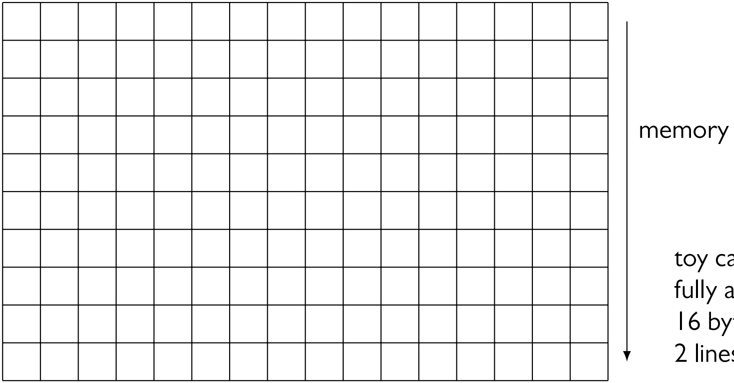
SRAM vs DRAM

- SRAM is faster but requires more transistors
- SRAM is used for caches, DRAM for main memory

Cache Levels

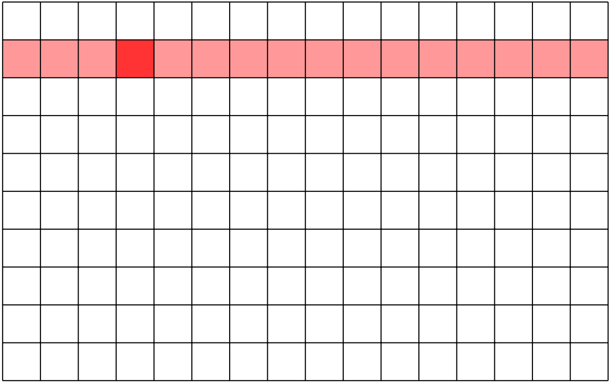
- Level 1, per-core, ∼3 cycles
- Level 2, per-core, ∼10 cycles
- Level 3, shared, ∼30-80 cycles
- eDRAM, shared, ∼200 cycles
- Memory, ∼300 cycles
- NUMA, ∼600-6000 cycles

Loading Memory into the Cache



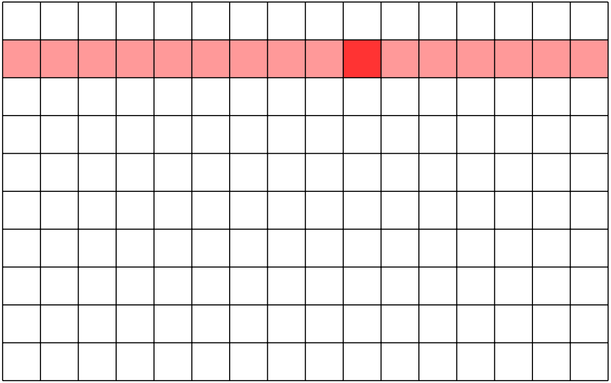
toy cache example
fully associative
16 bytes per line
2 lines of cache

Loading Memory into the Cache



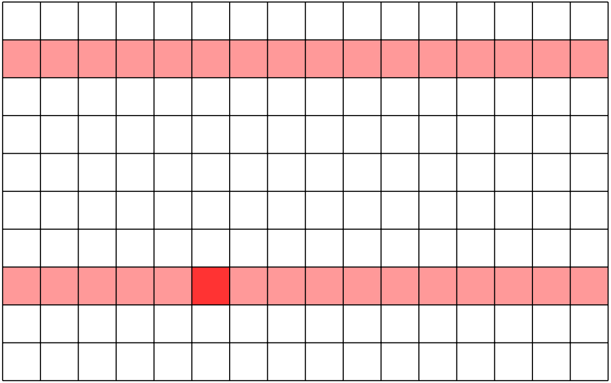
miss

Loading Memory into the Cache



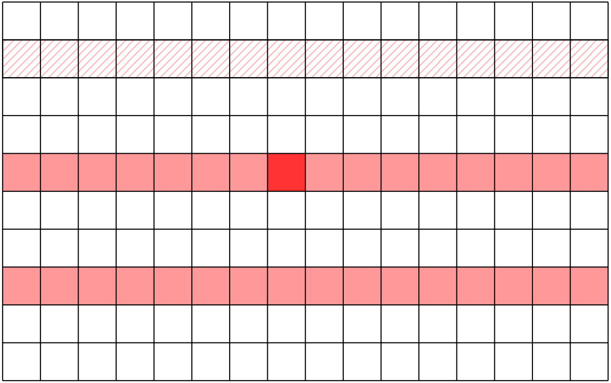
hit

Loading Memory into the Cache



miss

Loading Memory into the Cache



miss, old evicted

Cache Associativity

Memory to Cache Mapping

- Memory location mapped to cache locations (non-injective)
- Can be mapped to multiple locations

Fully Associative Memory can be mapped to any cache location

Direct Mapped Memory can be mapped to 1 cache location

N-way Associative Memory can be mapped to N cache locations

Making Good Use of the Cache

Spatial Locality

- Group together related pieces of data
- Minimize latency to obtain all pieces of information you need to compute things

Temporal Locality

- If you’re going to reuse data from a given location, do it before it gets evicted
- Ordering of memory access should prefer processing contiguous chunks linearly

Compact Layout

Hot Objects

- Keep hot objects small, get all data with few memory accesses
- If big enough, align on cache boundary
- Avoid wasted data due to padding

Local Storage

- Prefer storing data in the object itself
- Hold objects by value, avoid pointers to everywhere in memory

Impact of Padding

40 bytes, 3 cache lines, 10 wasted bytes

```
struct Person {
  enum : uint8_t { MALE, FEMALE, OTHER } sex;
  uint64_t id;
  char name[20];
  uint8_t age;
};
```

32 bytes, 2 cache lines, 2 wasted bytes

```
struct Person {
  uint64_t id;
  enum : uint8_t { MALE, FEMALE, OTHER } sex;
  uint8_t age;
  char name[20];
};
```

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Performance of Jumps

Taken

- Pre-emptively fetched the right location into the cache
- Pre-emptively started executing instructions in the pipeline
- No latency penalty

Not Taken

- Flush the pipeline and stall
- Even more costly if location not in cache or worse, not paged in
- Big penalty to both latency and throughput

Branchless Code

Avoiding Branches

- Compute all paths, then select result at the end
- Some instructions to do this natively, otherwise bitwise tricks
- Required for SIMD computation

Performance

- Good if all paths equally important with all similar amount of work
- Needs to be able to compute all paths regardless of condition
- Higher latency than a taken branch, typically good throughput

Static Branch Prediction

Heuristic for Cold State

- Only used if no historical data available
- Might not be used for modern processors
- Easy to annotate in source code (likely/unlikely)

Jump Direction

- Backward jump means loop, assume taken
- Forward jump means branch, assume not-taken
- Absolute jumps (virtual functions), assume nothing

Dynamic Branch Prediction

History Cache

- Store N previous states of the branch (e.g. 32)
- Assume that patterns repeat themselves
- Multiple branches might map to the same cache entry
  - Possibility for branches with same history slot to collide, causing spurious mispredictions
  - Some architectures mitigate this to some degree

Hotness

- Control flows that happen more often will be faster
- If some control flow you care about doesn't happen often, you need to make it so

Take Aways

- Hardware is smart, help it be good at what it’s trying to do
- Hierarchical parallelism, instruction and thread level
- Different instructions have different speed
- Memory and caches most important thing
    - Make your algorithms have good traversal orders
    - Design data structures with good object layout

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

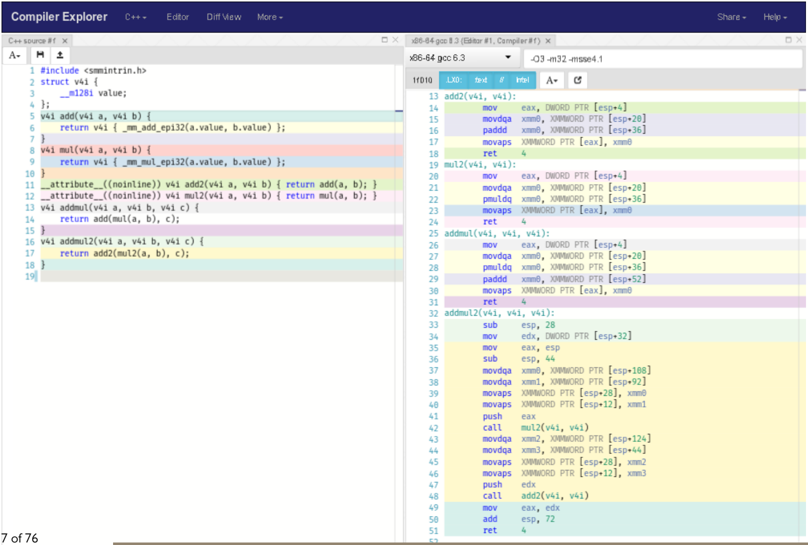
Inlining

Problem
Calling a function has some overhead
Pushing registers on stack, possibly a high cost if operating on registers
Jumping to another address

Solution
Expand the function at the call site
Also enables to optimize that call for that set of arguments

Drawbacks
More code, not benefitting from sharing the same instruction cache

Function Call Cost and ABI (1)



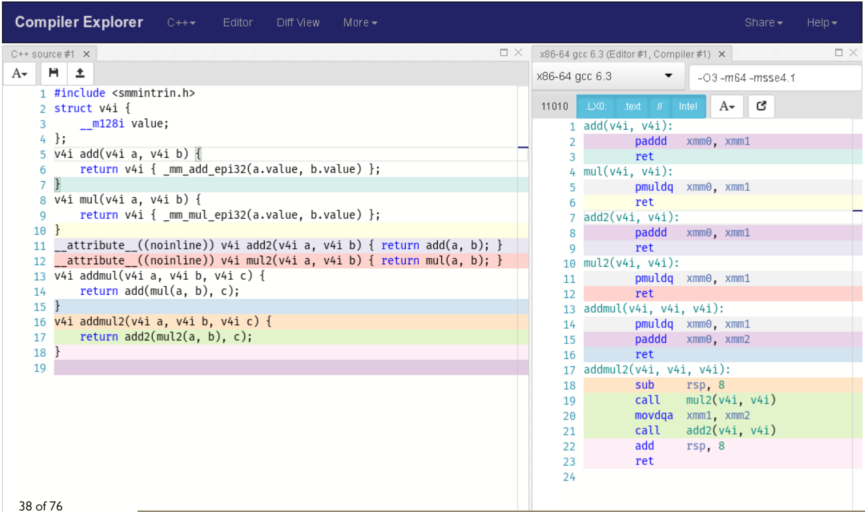
```cpp
#include <smmintrin.h>
struct v4i {
    __m128i value;
};
v4i add(v4i a, v4i b) {
    return v4i { _mm_add_epi32(a.value, b.value) };
}
v4i mul(v4i a, v4i b) {
    return v4i { _mm_mul_epi32(a.value, b.value) };
}
__attribute__((noinline)) v4i add2(v4i a, v4i b) { return add(a, b); }
__attribute__((noinline)) v4i mul2(v4i a, v4i b) { return mul(a, b); }
v4i addmul(v4i a, v4i b, v4i c) {
    return add(mul(a, b), c);
}
v4i addmul2(v4i a, v4i b, v4i c) {
    return add2(mul2(a, b), c);
}
```

x86-64 gcc 6.3

-O3 -m32 -msse4.1

```asm
add2(v4i, v4i):
        mov     eax, DWORD PTR [esp+4]
        movdqa  xmm0, XMMWORD PTR [esp+20]
        paddd   xmm0, XMMWORD PTR [esp+36]
        movaps  XMMWORD PTR [eax], xmm0
        ret     4
mul2(v4i, v4i):
        mov     eax, DWORD PTR [esp+4]
        movdqa  xmm0, XMMWORD PTR [esp+20]
        pmuldq  xmm0, XMMWORD PTR [esp+36]
        movaps  XMMWORD PTR [eax], xmm0
        ret     4
addmul(v4i, v4i, v4i):
        mov     eax, DWORD PTR [esp+4]
        movdqa  xmm0, XMMWORD PTR [esp+20]
        pmuldq  xmm0, XMMWORD PTR [esp+36]
        paddd   xmm0, XMMWORD PTR [esp+52]
        movaps  XMMWORD PTR [eax], xmm0
        ret     4
addmul2(v4i, v4i, v4i):
        sub     esp, 28
        mov     edx, DWORD PTR [esp+32]
        mov     eax, esp
        sub     esp, 44
        movdqa  xmm0, XMMWORD PTR [esp+108]
        movdqa  xmm1, XMMWORD PTR [esp+92]
        movaps  XMMWORD PTR [esp+28], xmm0
        movaps  XMMWORD PTR [esp+12], xmm1
        push    eax
        call    mul2(v4i, v4i)
        movdqa  xmm2, XMMWORD PTR [esp+124]
        movdqa  xmm3, XMMWORD PTR [esp+44]
        movaps  XMMWORD PTR [esp+28], xmm2
        movaps  XMMWORD PTR [esp+12], xmm3
        push    edx
        call    add2(v4i, v4i)
        mov     eax, edx
        add     esp, 72
        ret     4
```

Function Call Cost and ABI (2)



-O3 -m64 -msse4.1

```cpp
#include <smmintrin.h>
struct v4i {
    __m128i value;
};
v4i add(v4i a, v4i b) {
    return v4i { _mm_add_epi32(a.value, b.value) };
}
v4i mul(v4i a, v4i b) {
    return v4i { _mm_mul_epi32(a.value, b.value) };
}
__attribute__((noinline)) v4i add2(v4i a, v4i b) { return add(a, b); }
__attribute__((noinline)) v4i mul2(v4i a, v4i b) { return mul(a, b); }
v4i addmul(v4i a, v4i b, v4i c) {
    return add(mul(a, b), c);
}
v4i addmul2(v4i a, v4i b, v4i c) {
    return add2(mul2(a, b), c);
}
```

```asm
add(v4i, v4i):
        paddd   xmm0, xmm1
        ret
mul(v4i, v4i):
        pmuldq  xmm0, xmm1
        ret
add2(v4i, v4i):
        paddd   xmm0, xmm1
        ret
mul2(v4i, v4i):
        pmuldq  xmm0, xmm1
        ret
addmul(v4i, v4i, v4i):
        pmuldq  xmm0, xmm1
        paddd   xmm0, xmm2
        ret
addmul2(v4i, v4i, v4i):
        sub     rsp, 8
        call    mul2(v4i, v4i)
        movdqa  xmm1, xmm2
        call    add2(v4i, v4i)
        add     rsp, 8
        ret
```

Specialization

Problem Some functions need to work with various different kinds of input
Supporting all cases it is slow

Solution Generate different versions of the function optimized for special cases

Drawbacks More code, not benefitting from sharing the same instruction cache
Meta-programming and code generation techniques add extra engineering overhead

What to Specialize

- Inline to automatically enable more specialization opportunities for a given set of arguments
- Generate versions of algorithms for different data sizes, e.g. Fast Fourier Transform optimized for working with 4096 or 8192 samples... Can pick on entry with a switch.
- Static types instead of boxed dynamic ones
- Branches on the outside of loops rather than inside

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Scalar Promotion

Problem
Memory access is expensive
Most code has implicit memory accesses everywhere
Compiler cannot always perform the necessary alias analysis to optimize them away

Solution
Explicitly perform loads and stores
Remove access to invariant data out of loops

Drawbacks
Additional registers required to maintain the data as opposed to fetching it whenever needed

Aliasing

```
void foo(int* array, int const& size, int const& value) {
    for (int i=0; i<size; ++i)
        array[i] = 2 * value;
}
```

```
void foo(int* array, int const& size, int const& value) {
    int sz = size;
    int v = value;
    for (int i=0; i<sz; ++i)
        array[i] = 2 * v;
}
```

Unrolling

Problem Comparing and jumping at each iteration is slow

Solution Treat more than one element per iteration

Drawbacks Larger code
Need to deal with data sizes not dividable by the unrolling size

Unrolling with C++ Templates

```
int i;
for(i=0; i<n/4*4; i+=4) {
  f(i);
  f(i+1);
  f(i+2);
  f(i+3);
}
for(; i<n; ++i)
  f(i);
```

```
unroll<4>(
  0, n,
  [&](int i) {
    f(i);
  }
)
```

Pipelining and Streaming

Problem Chain of dependent operations prevent parallel execution

Solution Organize the operations in iterations and make every operation in the chain consume the output of previous iterations

Drawbacks Higher memory requirements (buffers, registers)
More complicated to set up

Streaming Computation to Another Device

Dumb Solution

- Send data to device
- Process data on device
- Receive result back

Pipelined Solution

Send block 0		
Send block 1	Process block 0	
Send block 2	Process block 1	Receive block 0
Send block 3	Process block 2	Receive block 1
	Process block 3	Receive block 2
		Receive block 3

Size of block must be well chosen to overlap communication and computation

Pipeling Loops

Some Chain in a Loop

```
for(size_t i=0; i!=n; ++i)
  d(c(b(a(i))));
```

Operation Latencies

- a, b and d: 3
- c: 6

Unrolling-based Pipelining

```
for(size_t i=0; i!=n; i += 6) {
  r0 = a(i);
  r1 = a(i+1);
  r2 = a(i+2);
  r3 = a(i+3); r0 = b(r0);
  r4 = a(i+4); r1 = b(r1);
  r5 = a(i+5); r2 = b(r2);
               r3 = b(r3); r0 = c(r0);
               r4 = b(r4); r1 = c(r1);
               r5 = b(a4); r2 = c(r2);
                           r3 = c(r3);
                           r4 = c(r4);
                           r5 = c(r5);
                                       d(r0);
                                       d(r1);
                                       d(r2);
                                       d(r3);
                                       d(r4);
                                       d(r5);
}
```

$\max_i$ *$latency_i$* registers and unrolling size

Register Rotation

```
r0 = a(0);
r1 = a(1);
r2 = a(2);
r3 = a(3); r0 = b(r0);
           r1 = b(r1);
           r2 = b(r2);
                       r0 = c(r0);
                       r1 = c(r1);

for(size_t i=0; i!=n-4; ++i) {
  r4 = a(i+4); r13 = b(r3); r12 = c(r2); d(r0);

  r0 = r1;
  r3 = r4; r2 = r13; r1 = r12;
}
```

tight code with no unrolling

$\sum_i \frac{latency_i}{steps-1}$ registers

Tiling

Problem	Traversing large arrays might cause redundant access to main memory
Solution	Organize the array in smaller tiles that fits in the cache
Drawbacks	Optimal size of the tile depends on the actual hardware

2D Traversal: the Basics

Bad

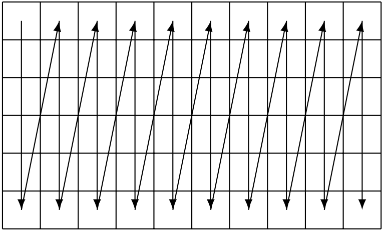
```
for (size_t i=0; i<numcols; ++i)
   for (size_t j=0; j<numrows; ++j)
      table[j*numrows + i] *= 2;
```

Better

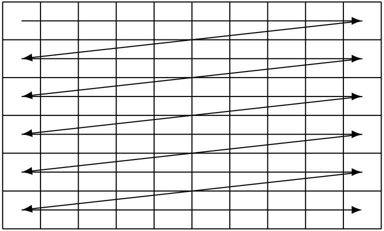
```
for (size_t j=0; j<numrows; ++j)
  for (size_t i=0; i<numcols; ++i)
      table[j*numrows + i] *= 2;
```

Simple 2D Traversal

Vertical (Useless)

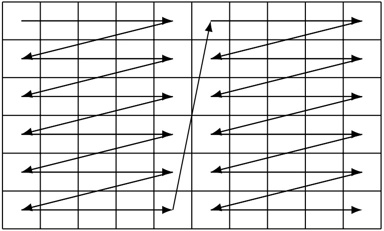


Horizontal (Element-Wise Operations)

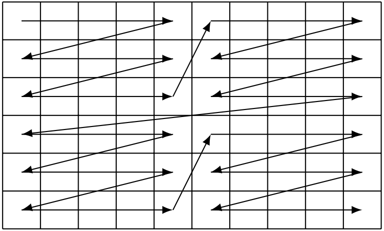


Tiled 2D Traversal

Cache-Aware Vertical (Vertical Reductions)



Tiles (Stencils, Linear Algebra...)



Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Instruction Cache

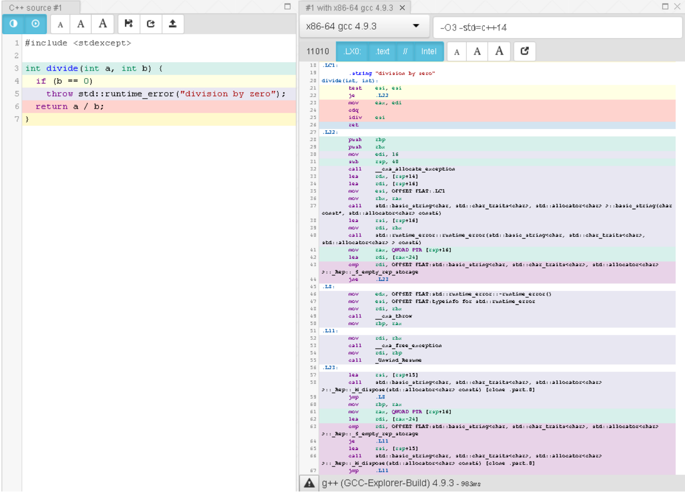
Code Locality

- It's better if code is tight and fits in the cache
- Code that's not hot does not need to waste space in your cache lines
- Optimizing for cache typically conflicts a lot with other optimizations, hard to do

Hot vs Cold

- Group together the parts of the code that needs to be fast
- Move the exceptional paths far away
- Compilers can only move paths the end of the function, to make things be further away you need to move it to another function or section

Exceptions and Codegen (I)



x86-64 gcc 4.9.3

-O3 -std=c++14

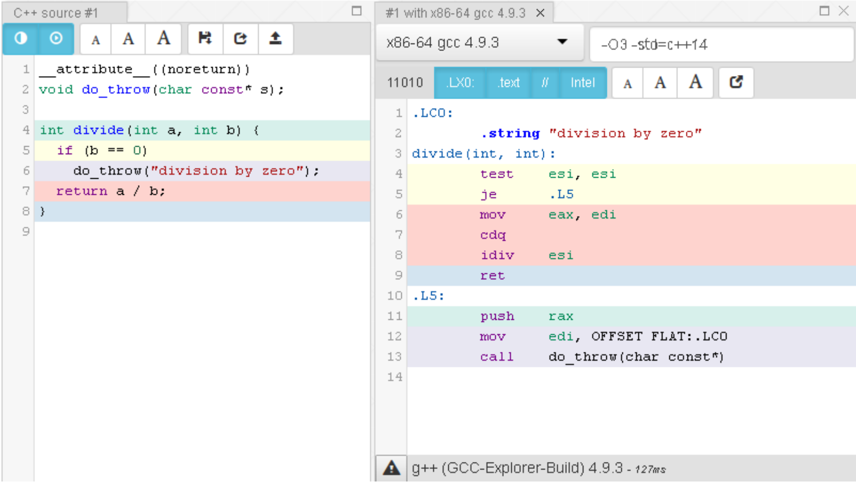
```cpp
#include <stdexcept>

int divide(int a, int b) {
  if (b == 0)
    throw std::runtime_error("division by zero");
  return a / b;
}
```

```asm
.LC1:
        .string "division by zero"
divide(int, int):
        test    esi, esi
        je      .L22
        mov     eax, edi
        cdq
        idiv    esi
        ret
.L22:
        push    rbp
        push    rbx
        mov     edi, 16
        sub     rsp, 40
        call    __cxa_allocate_exception
        lea     rdx, [rsp+14]
        lea     rdi, [rsp+16]
        mov     esi, OFFSET FLAT:.LC1
        mov     rbx, rax
        call    std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string(char
const*, std::allocator<char> const&)
        lea     rsi, [rsp+16]
        mov     rdi, rbx
        call    std::runtime_error::runtime_error(std::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&)
        mov     rax, QWORD PTR [rsp+16]
        lea     rdi, [rax-24]
        cmp     rdi, OFFSET FLAT:std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_Rep::_S_empty_rep_storage
        jne     .L23
.L8:
        mov     edx, OFFSET FLAT:std::runtime_error::~runtime_error()
        mov     esi, OFFSET FLAT:typeinfo for std::runtime_error
        mov     rdi, rbx
        call    __cxa_throw
        mov     rbp, rax
.L11:
        mov     rdi, rbx
        call    __cxa_free_exception
        mov     rdi, rbp
        call    _Unwind_Resume
.L23:
        lea     rsi, [rsp+15]
        call    std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_Rep::_M_dispose(std::allocator<char> const&) [clone .part.0]
        jmp     .L8
        mov     rbp, rax
        mov     rax, QWORD PTR [rsp+16]
        lea     rdi, [rax-24]
        cmp     rdi, OFFSET FLAT:std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_Rep::_S_empty_rep_storage
        je      .L11
        lea     rsi, [rsp+15]
        call    std::basic_string<char, std::char_traits<char>, std::allocator<char>
>::_Rep::_M_dispose(std::allocator<char> const&) [clone .part.0]
        jmp     .L11
```

g++ (GCC-Explorer-Build) 4.9.3 - 983ms

Exceptions and Codegen (2)



```cpp
__attribute__((noreturn))
void do_throw(char const* s);

int divide(int a, int b) {
  if (b == 0)
    do_throw("division by zero");
  return a / b;
}
```

x86-64 gcc 4.9.3

-O3 -std=c++14

```
.LC0:
        .string "division by zero"
divide(int, int):
        test    esi, esi
        je      .L5
        mov     eax, edi
        cdq
        idiv    esi
        ret
.L5:
        push    rax
        mov     edi, OFFSET FLAT:.LC0
        call    do_throw(char const*)
```

g++ (GCC-Explorer-Build) 4.9.3 - 127ms

Outline

Introduction

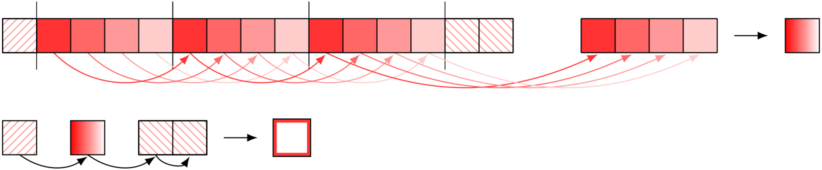
Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Vectorized Sum Algorithm



- Compute 4 adjacent sums in parallel
- Reduce the 4 partial sums to a single one
- Add with leading and trailing data to get result.

Vectorizing with Intrinsics

Scalar code

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    uint32_t result = 0;
    for(; first != last; ++first)
        result += *first;
    return result;
}
```

SSE2 Intrinsics

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
  __m128i result = _mm_set1_epi32(0);
  for(; first != last; first += 8)
  {
    __m128i in = _mm_load_si128((__m128i*)first);
    __m128i lo = _mm_unpacklo_epi16(in, _mm_setzero_si128());
    __m128i hi = _mm_unpackhi_epi16(in, _mm_setzero_si128());

    result = _mm_add_epi32(result, lo);
    result = _mm_add_epi32(result, hi);
  }

  result = _mm_add_epi32(result,
    _mm_shuffle_epi32(result, _MM_SHUFFLE(1, 0, 3, 2))
  );
  result = _mm_add_epi32(result,
    _mm_shuffle_epi32(result, _MM_SHUFFLE(0, 1, 2, 3))
  );
  return _mm_cvtsi128_si32(result);
}
```

C++ Libraries for SIMD computing

Promote to 32-bit on load

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint32_t> out_t;
    out_t result = 0;
    for(; first != last; first += out_t::size())
    {
        result += out_t(first);
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Load as 16-bit and explicitly promote

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    typedef datapar<uint16_t> in_t;
    typedef datapar<uint32_t, in_t::size()/2> out_t;
    out_t result = 0;
    for(; first != last; first += in_t::size())
    {
        in_t in(first);
        auto [lo, hi] = datapar_cast<out_t>(in);
        result += lo;
        result += hi;
    }
    return result.reduce(
        [](auto a, auto b) { return a + b; }
    );
}
```

Vc by Matthias Kretz
Boost.SIMD by Joel Falcou et. al
ISO C++ TS Parallelism v2

Outline

Introduction

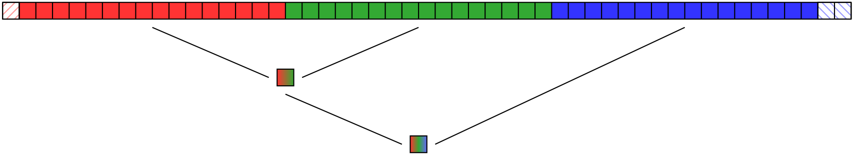
Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

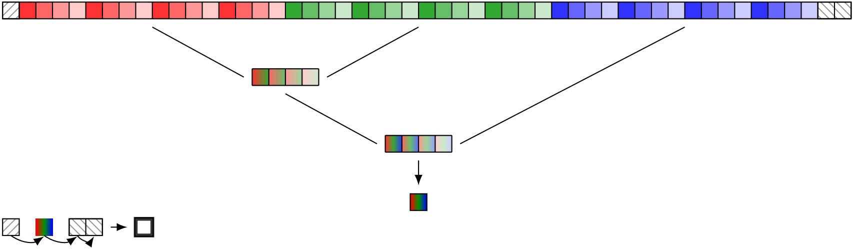
- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Generalizing a Multi-Core Sum



- Split range into a number of subranges that get processed independently
- Preferably give full cache lines to each thread to avoid false sharing
- Accumulate the results pairwise to avoid global synchronization

Multi-Core SIMD Reduction



- Combine the two approaches
- No need for each thread to deal with leading/trailing data if splitting on cache line boundary

Parallel Algorithm Skeletons

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(datapar, first, last,
        overload(
            [](uint16_t a, uint16_t b)
            {
                return a + b;
            },
            [](datapar<uint16_t> a, datapar<uint16_t> b)
            {
                return datapar<uint32_t, a.size()>(a)
                     + datapar<uint32_t, b.size()>(b);
            }
        )
    );
}
```

- Re-usable SIMD-enabled skeleton
- Handles leading and trailing data based on scalar overload
- Handles SIMD vector to scalar reduction with $log_2(N)$ algorithm

Parallel Algorithm Skeletons

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(par_datapar, first, last,
        overload(
            [](uint16_t a, uint16_t b)
            {
                return a + b;
            },
            [](datapar<uint16_t> a, datapar<uint16_t> b)
            {
                return datapar<uint32_t, a.size()>(a)
                     + datapar<uint32_t, b.size()>(b);
            }
        )
    );
}
```

- Re-usable SIMD-enabled skeleton
- Handles leading and trailing data based on scalar overload
- Handles SIMD vector to scalar reduction with $log_2(N)$ algorithm
- Also parallelized on multi-core

Parallel Algorithm Skeletons, Polymorphic

```
uint32_t sum(uint16_t* first, uint16_t* last)
{
    return std::reduce(datapar, first, last,
        [](auto a, auto b)
        {
            return cast<uint32_t>(a) + cast<uint32_t>(b);
        }
    );
}
```

- C++14 polymorphic lambdas allow to express a single generic overload for both scalar and SIMD
- Goal is to make it possible to write SIMD-agnostic code as much as possible.

Higher-Order Algorithms

Classic Data Parallel Skeletons

- map, `std::transform`
- `fold, std::reduce`
- `scan, std::inclusive_scan or std::exclusive_scan`

Task Parallel Skeletons

- graph of tasks
- farms
- pipelines
- wavefronts

Why Think in Skeletons

Code Speed

- Well-studied patterns, good algorithms in research
- Plenty of tuned implementations
- Composable for layers of backends

Productivity Speed

- Embed optimization techniques for a given problem in a box
- Reuse box with minimal extra work
- Abstraction decouples the algorithm from the optimization

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Array-of-Structures vs Structures-of-Arrays

AoS

```
struct Sample
{
  double portfolio_value;
  double external_flow;
};

double time_weighted_return(vector<Sample> const& samples)
{
  double ret = 1.;
  for(size_t i=1; i<samples.size(); ++i)
  {
    ret *= (samples[i].portfolio_value - samples[i].external_flow)
         / samples[i-1].portfolio_value;
  }
  return ret;
}
```

SoA

```
struct Samples
{
  vector<double> portfolio_values;
  vector<double> external_flows;
};

double time_weighted_return(Samples const& samples)
{
  double ret = 1.;
  for(size_t i=1; i<samples.portfolio_values.size(); ++i)
  {
    ret *= (samples.portfolio_values[i] - samples.external_flows[i])
         / samples.portfolio_values[i-1];
  }
  return ret;
}
```

- AoS more natural, humans organize things into objects
- SoA makes it easier to vectorize, contiguous memory makes load/store easy
- AoS is more compact as it puts things into the same cache line
  - usually translates to better latency but worse throughput

Outline

Introduction

Part 1: Understanding the Hardware

- Instruction-Level Parallelism
- Thread-Level Parallelism
- Caches
- Branch Prediction

Part 2: Hardware-Aware Patterns

- Propagation, Expansion and Specialization
- Data Access Patterns
- Instruction Cache
- Vectorization
- High-level Parallelization
- Object Layout
- Computation Shortcuts

Strength Reduction

Replace operations by cheaper ones

- Multiplication by addition
- Exponentiation by multiplication
- Division by multiplication by reciprocal

Linearization of Multi-Dimensional Access

```
template<class T>
struct Matrix
{
  Matrix(size_t width, size_t height)
    : data(static_cast<T*>(operator new(width*height)))
    , width(width)
    , height(height)
  {}

  T* data;
  size_t width;
  size_t height;

  T& operator()(size_t j, size_t i) {
    return data[j*width+i];
  }
};
```

Linearization of Multi-Dimensional Access (2)

```
template<class T>
Matrix<T> identity(size_t width, size_t height)
{
  Matrix<T> m(width, height);
  for (size_t j=0; j<m.height; ++j) {
    for (size_t i=0; i<m.width; ++i)
      m(j, i) = 0.0;
    m(j, j) = 1.0;
}
```

Linearization of Multi-Dimensional Access (3)

```
template<class T>
Matrix<T> identity(size_t width, size_t height)
{
  Matrix<T> m(width, height);
  for (size_t i=0; i<width*height; ) {
    data[i++] = 1.0;
    for (size_t j=i+width; i<j; ++i)
      data[i] = 0.0;
  }
  return m;
}
```

Newton-Rhapson Method

- Use ”some way” to get an initial approximation y_0 of a function $f(x)$
- Find $g(y)$ so that $g(f(x)) = 0$
- Refine with

$$y_{i+1} = y_i - \frac{g(y_i)}{g'(y_i)}$$

- Repeat until happy

Common Approximations

Reciprocal $1/x$

- Dedicated instructions for y_0
- Refine with $y_{i+1} = y_i + y_i(1 - xy_i)$
- Can be used to implement fast division

Inverse Square Root $1/sqrt(x)$

- Quake III method 0x5f3759df or dedicated instructions for y_0
- Refine with $y_{i+1} = y_i(1.5 - 0.5 * x * y_i * y_i)$

Mixed Precision Techniques

Idea

- Convert input from double to single precision
- Compute result in single precision
- Convert it back to double precision
- Refine iteratively

Application

- Single precision is much faster on GPU
- Successfully deployed for some linear algebra problems

Fast Polynomial Evaluation

Polynoms

- Functions approximated by polynoms by intervals
- $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$

Horner

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + x a_4)))$$

```
r = a4;
r = fma(r, x, a3);
r = fma(r, x, a2);
r = fma(r, x, a1);
r = fma(r, x, a0);
```

Optimal number of additions and multiplications, which can be fused

Estrin’s Scheme

$$p(x) = (a0 + a_1 x) + (a_2 + a_3 x)x^2 + a_4 x^4$$

- Not as efficient, but can run multiple fmas in parallel
- Better utilisation of superscalar processors
- Only performs well for certain sizes

Questions?